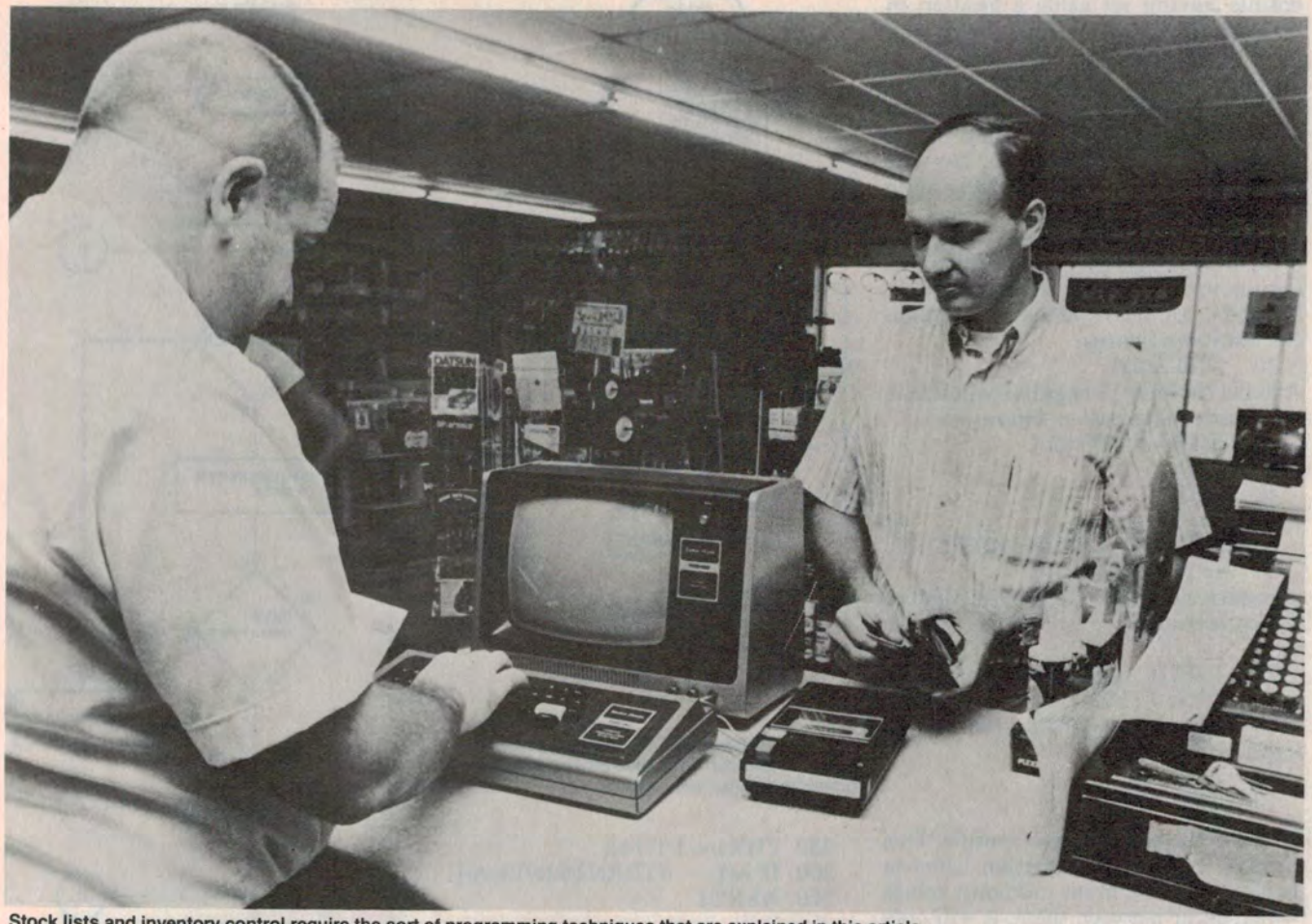


Stock lists and inventory control require the sort of programming techniques that are explained in this article.

Advanced BASIC

Phil Cohen

If you have followed the previous series on Back Door Into BASIC, or picked up the rudiments of BASIC programming elsewhere, you're probably asking yourself "where do I go from here?"

THE EASY ANSWER to that question in the introduction is — programming experience will reveal all to the initiate. But really, that just isn't true. There's a lot to be learned about programming which most people (myself included) could never develop from scratch — novel concepts and systems, applied mathematical methods et cetera.

These Advanced BASIC articles, which will appear in ETI from time to time, are intended to provide food for thought and to stimulate those who have exhausted their interest in computer games.

The language used is an 8K Microsoft BASIC, and minor alteration will allow the included examples to run on almost any medium-sized personal computer system.

Although the various parts of this series have been written to cover a particular concept or field of study, these are not covered exclusively — each part introduces other facets which add together to provide a useful program or major program segment in each part of the series.

Sorting

A very common problem in computing is sorting of one type or another. This can be anything from simple alphabetic sorting to sorting of time-dependent data from different sources to give an overall picture. There have been *books* written on different sorting algorithms — but we're not going into it quite as deeply as that!

This article covers three types of sorting: 1) pigeon-hole, 2) push-down and 3) ripple.

Pigeon-hole sorting

This is the fastest and most wasteful in memory space of the three types. It can only be used in some instances where the minimum difference between successive sorted data is fixed and known, and where the range of input values is also known. Essentially, it ▶

entails having set aside a location in memory for each possible input and putting the incoming data into its reserved location as it comes in.

This type of sort is useful for applications such as the handling of monthly sales figures — it has the additional advantage of reserving space for interpolated data.

As an example of the use of this type of algorithm, the following program will take in up to 52 weekly figures (weekly sales, for example) and then interpolate the unknown figures.

```
10  DIM A(52)
```

A holds the data. A negative value in A will mark an 'unknown' figure, so:

```
20  FOR I = 1 TO 52
30  A(I) = -1
40  NEXT I
```

Now input the data:

```
50  INPUT "WEEK NUMBER "; N
60  IF N < 0 THEN 105
```

A negative number will end the input of data and cause the results to be printed out.

```
70  INPUT "SALES "; S
80  IF S < 0 THEN 70
```

This stops locations being marked as unknown by mistake.

```
90  A(N) = S
100 GOTO 50
```

Now for the interpolation routine. This uses a linear approximation between two values to fill in the unknown values which occur between them. It will only do so between the earliest and latest known data.

```
105 J = 1
110 J = J + 1
120 IF J > 51 THEN 250
130 IF A(J) > -.5 THEN 110
```

The above will find an 'unknown' week. If there are none left, it will jump to 250, which prints the results. Now find the nearest known figures before and after point J (the unknown week which we found above):

```
150 FOR AF = J + 1 TO 52
160 IF A(AF) > -.5 THEN 190
170 NEXT AF
180 GOTO 110
190 FOR BF = J - 1 TO 1 STEP -1
200 IF A(BF) > -.5 THEN 230
210 NEXT BF
220 GOTO 110
```

BF and AF now contain the nearest 'known' weeks before and after J, respectively. Now interpolate:

```
230 A(J) = (A(BF)*(AF-J) +
      A(AF)*(J-BF)) / (AF-BF)
```

(This is a standard linear interpolation equation, which can be derived with a bit of geometry). Find the next unknown:

```
240 GOTO 110
```

Now print the results when finished:

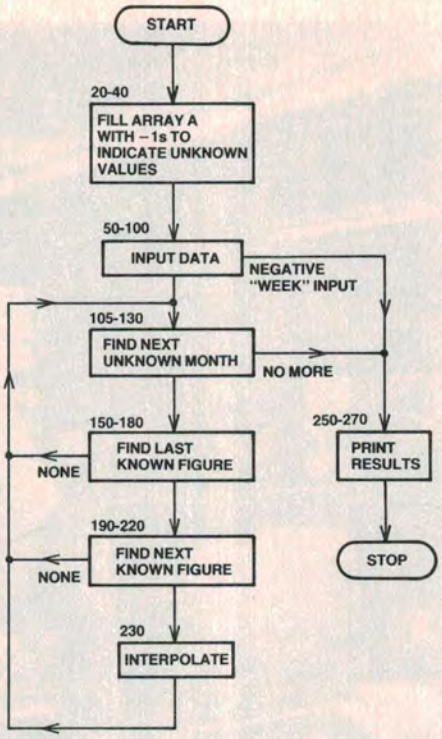


Figure 1. **A 'pigeon-hole' sorting routine with interpolation. This can be used to provide an educated guess at missing sales figures.**

```
250 FOR I = 1 TO 52
260 IF A(I) > -.5 THEN PRINT I, A(I)
270 NEXT I
```

A flowchart for the above program is given in Figure 1.

Push-down sorting

In applications where a variable number of data points is to be sorted and stored in order, push-down order is often the best course, being fairly simple to implement. The idea is that a list of the items in order is kept in a 'stack' structure. This is simply an array larger than the largest number of data to be held. The data is stored in the array elements with subscripts *below* a certain value. This value increases as more data is added.

The value of the subscript at the 'top' of the stack is held in the 'stack pointer'. This is incremented as data is added to the stack.

The addition of an item of data *in the middle of the stack* means that the rest of the data has to be 'pushed down' to accommodate it.

The following example will sort numbers into ascending order using a push-down algorithm (see Figure 2 for flowchart).

```
10  DIM S(50)
```

S is the stack — up to 50 items can be stored.

```
20  SP = 0
```

SP, the stack pointer, points to the highest location which is in use. Setting it to 0 indicates that the stack is empty.

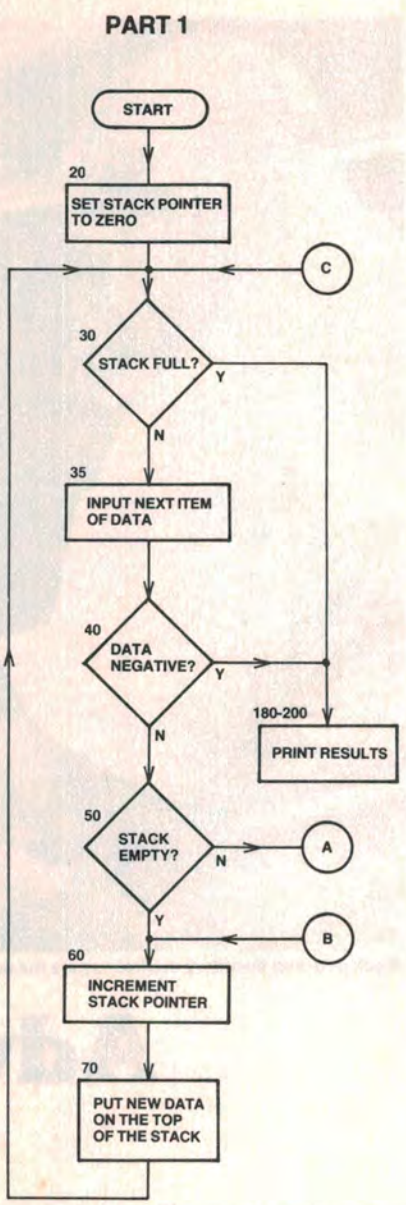


Figure 2. **Push-down sorting — full explanation in the text.**

```
30  IF SP = 50 THEN PRINT
      "NO ROOM" : GOTO 180
35  INPUT "DATA"; D
40  IF D < 0 THEN 180
```

inputs the data (which is assumed to be above zero). A number below 0 is taken as an instruction to print the results. Line 30 checks to see if the stack is full up. First, find out where to put D — check to see if it's the *only* data:

```
50  IF SP > 0 THEN 90
60  SP = SP + 1
70  S(SP) = D
80  GOTO 30
```

If it's not the *only* data, find out where it should be in the stack:


```
90   FOR I = 1 TO SP
100  IF S(I) > D THEN 130
110  NEXT I
```

If D reaches this point then it must be the largest item, so

```
120  GOTO 60
```

puts it on the 'top' of the stack. If line 100 sends it to line 130, it must come just before item I. Push the data down just before item I:

```
130  FOR J = SP TO I STEP -1
140  S(J+1) = S(J)
150  NEXT J
```

and then insert D:

```
160  S(I) = D
165  SP = SP + 1
170  GOTO 30
```

Now print the results:

```
180  FOR I = 1 TO SP
190  PRINT S(I)
200  NEXT I
```

Ripple sort

This is probably the best-known sorting algorithm. It is also known as bubble sort or even, on occasion, travelling-wave sort.

The way it works is this: the data is put into a fixed-length array (or, as in the case of the example to follow, a stack). Starting at one end, the program compares successive pairs of items and swaps their positions in the list if they appear in the wrong order. It repeats this, starting at the same end each time, until it hasn't made any changes in the latest pass. Thus a 'bubble' of change sweeps up through the data array. ▶

The following program allows named data items with several parameters to be stored or deleted one at a time. They can also be sorted according to any one of the parameters. A full flowchart is given in Figures 3 to 6.

```
100 REM MAIN PROGRAM
110 INPUT "HOW MANY ITEMS
    (MAXIMUM) "; NI
120 INPUT "HOW MANY
    VARIABLES PER ITEM "; NV
130 DIM A(NI, NV), A$(NI), N$ (NV)
```

A holds the parameters associated with each item, A$ holds the item names and N$ holds the parameter type names.

```
140 FOR I = 1 TO NV
150 PRINT "WHAT IS
    VARIABLE ";I;" CALLED ";
160 INPUT N$ (I)
165 NEXT I
170 SP = 0
```

SP is the stack pointer.

```
180 INPUT "COMMAND "; C$
190 IF C$ = "I" THEN GOSUB 1000
200 IF C$ = "S" THEN GOSUB 2000
210 IF C$ = "R" THEN GOSUB 3000
220 IF C$ = "E" THEN STOP
230 GOTO 180
```

inputs the command and takes the appropriate action. I = input, S = sort, R = remove and E = end.

```
1000 REM INPUT
1020 IF SP › = NI THEN PRINT
     "NO ROOM" : RETURN
```

checks for stack overflow.

```
1030 SP = SP + 1
1040 INPUT "ITEM NAME "; A$(SP)
1050 FOR I = 1 TO NV
1060 PRINT "WHAT IS THE ";
     N$(I); " OF ITEM "; A$(SP)
1070 INPUT A(SP, I)
1080 NEXT I
1090 RETURN
```

The above section of code inputs the new item and its parameters and puts them on to the end of the stack, incrementing the stack pointer.

```
2000 REM SORT
2005 IF SP ‹ 2 THEN PRINT "NOT
     ENOUGH ITEMS" : RETURN
```

Stops the user trying to sort one item! Trying to sort one item will upset the algorithm used.

```
2010 INPUT "SORT ACCORDING
     TO WHAT "; S$
2020 FOR I = 1 TO NV
2030 IF N$(I) = S$ THEN 2060
2040 NEXT I
2050 PRINT
     "NOT FOUND ": RETURN
```

finds out which parameter to use in the sort.

```
2060 S = I
```

stores the result of the above so that we can use I for the next loop (it's traditional to use the letter I for this because in FORTRAN it represents the first integer variable).

```
2070 F = 0
```

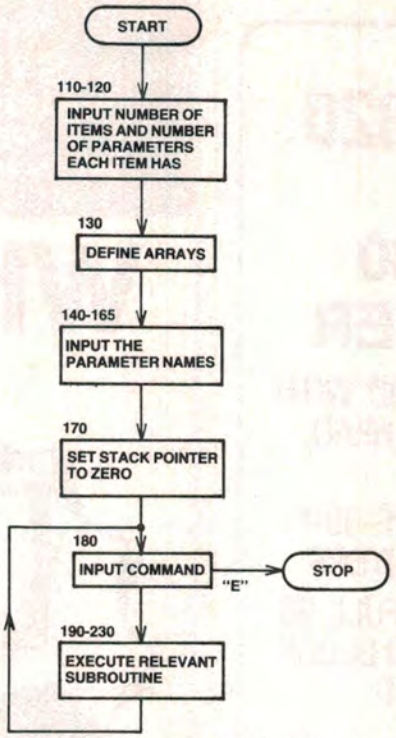


Figure 3. **The main program. This calls all the subroutines and initialises everything.**

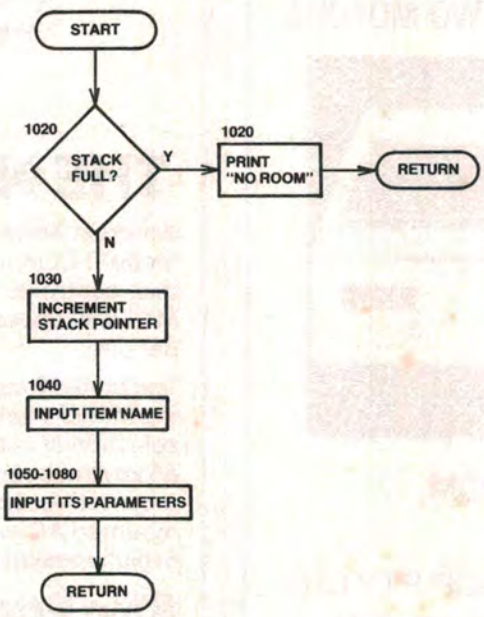


Figure 4. **Adding an item.**

F is a flag which indicates (if it is set to 1) that a swap has been made on the latest pass.

```
2080 FOR I = 2 TO SP
2090 IF A(I, S) › = A(I – 1, S) THEN
     2190
```

If the two items don't need to be swapped, line 2090 skips the next bit.

```
2100 T$ = A$(I)
2110 A$(I) = A$(I – 1)
2120 A$(I – 1) = T$
2130 FOR J = 1 TO NV
2140 T = A(I, J)
```

PART 1

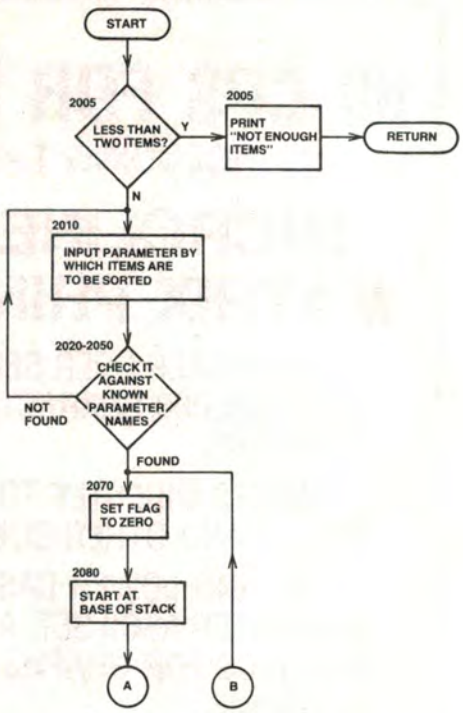
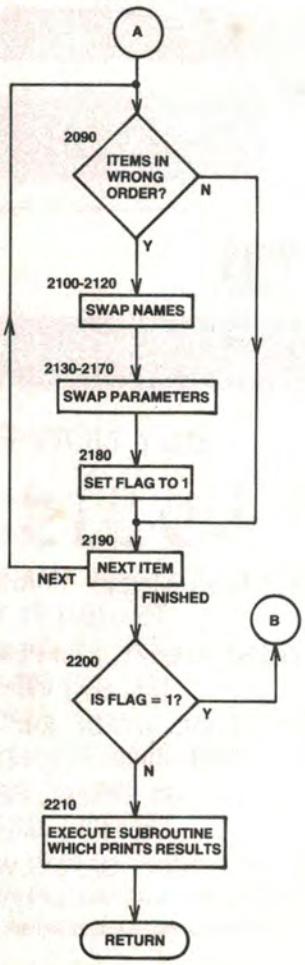


Figure 5. **The sorting routine itself.**

PART 2



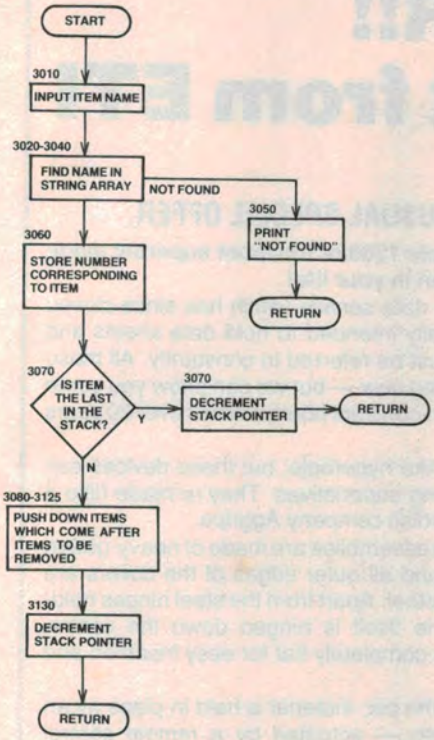


Figure 6. **Removing an item. This is fairly similar to Figure 2 in parts.**

```
2150  A(I, J) = A(I – 1, J)
2160  A(I – 1, J) = T
2170  NEXT J
2180  F = 1
```

swaps the two items (including their parameters). T$ and T are temporary stores. F is set to 1 to show that a change has been made in the current pass.

```
2190  NEXT I
```

tests the next pair.

```
2200  IF F = 1 THEN 2070
```

repeats the whole thing if there have been any swaps. Hopefully, after enough 'ripples', the program will do one pass without finding *any* items in the wrong order.

```
2210  GOSUB 4000
```

prints the results.

```
2220  RETURN
3000  REM REMOVE
3010  INPUT "WHICH ITEM "; S$
3020  FOR I = 1 TO SP
3030  IF A$(I) = S$ THEN 3060
3040  NEXT I
3050  PRINT
        "NOT FOUND" : RETURN
```

finds out which item is to be removed. Store I (as in line 2060):

```
3060  S = I
```

Check to see if the item is at the end of the stack:

```
3070  IF S = SP THEN
        SP = SP – 1 : RETURN
```

deals with this case.

The next section is essentially a push-down sort in reverse, the only complication being the moving of the parameter values. It might be instructive to look at the differences between Figures 2 and 6.

```
3080  FOR I = S to SP
3090  A$(I) = A$(I + 1)
3100  FOR J = 1 TO NV
3110  A(I, J) = A(I + 1, J)
3120  NEXT J
3125  NEXT I
3130  SP = SP – 1
3140  RETURN
```

The next routine prints out the list. All the string variables are limited to eight characters for format reasons. This routine is only entered via the sort routine.

```
4000  REM PRINT
4020  PRINT
4030  PRINT "NAME",
4040  FOR I = 1 TO NV
4050  PRINT LEFT$ (N$ (I), 8),
4060  NEXT I
4070  PRINT
4080  FOR I = 1 TO SP
4090  PRINT LEFT$ (A$ (I), 8),
4095  FOR J = 1 TO NV
4100  PRINT A(I, J),
4110  NEXT J
4120  PRINT
4130  NEXT I
4140  RETURN
```